SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 8, 257–267 (1996)

Practice

Live *Short* and Prosper: Applying Cost Analysis to Routine Software Maintenance Procedures

VLADIMIR M. TSIVKIN
7 Rosano Road, Stamford, CT 06905, U.S.A.

SUMMARY

Purely theoretical models of the software development and maintenance processes enjoy little use in the practical world; 'gut feeling' and hunch enjoy a large following when cost-effectiveness is the concern. This paper provides the calculations necessary to compare two methods (an 'all-in-one' method and an 'incremental' method) of transferring a project to quality assurance (QA). This paper compares the results of these calculations with the costs of doing two actual projects, one with each method. Both the calculations and the real projects indicate that QA costs are less when using the 'all-in-one' method. Since the calculations reasonably anticipated the real project experience, this paper argues that making such calculations has value when choosing between alternative processes. Such calculations are not purely theoretical, but can be used by managers to save money in software development and maintenance.

KEY WORDS: software release; software version; software cost experience; software cost estimation; software quality assurance; software maintenance

1. INTRODUCTION

In the business world of software development, theory and practice are traditionally incompatible where decisions related to the software development life cycle (SDLC) or the software maintenance life cycle (SMLC) are concerned. Practical decisions are often based on 'gut feelings' or what feels right to the people involved. Formula-based decisions remain both too abstruse and aloof from the routine real-world environment to be of much use. Moreover, so-called experience-oriented analyses rooted in practice, tend to describe all but how to apply their results successfully to situations outside of the one discussed. Few formula-based examinations of the life cycle zero in on decision-making, and how to make the life cycle processes less expensive and more reliable (Boehm, 1981, part II).

This paper demonstrates the practical advantages of applying a simple cost-analysing formula to a management choice situation that is common in all software development and maintenance life cycles that include some quality assurance (QA) processes (Boehm, 1981, part IIIA). For our purposes here and as is usually the case, software maintenance

CCC 1040–550X/96/040257–11

Received 22 August 1995
Revised 12 March 1996

(the modification, customization, and correction of already existing software) is presumed to be performed more often and require more resources than software development (Arthur, 1988, chapter 1, page 229; Martin and McClure, 1983, chapter 1).

Perhaps the most nebulous parts of the SDLC/SMLC occur during the transitions from any phase followed by a software quality assurance (QA) phase or stage, and especially the one done just before the release of the software for customer use. The QA processes in such a QA phase or stage often include but are not limited to unit testing and integration testing (Harrold and Soffa, 1988; McManus, 1987). Regression testing is often also included when software versioning is done, as well as being a normal part of the SMLC (Chapin, 1988; Rothermel and Harrold, 1993).

2. INTERFACE BETWEEN MAINTENANCE AND QA

Maintaining software application quality implies that no regression occurs after features have been changed in or added to the existing software package. The practical response traditionally has been to advocate that an application should be carefully tested after changes have been embedded (Osborne, 1985). An associated practical question is, how should the software be turned over to QA? Is there more than one way to accomplish such a turnover (McManus, 1987)?

In fact, many different interfaces between Maintenance and QA in a firm or organization ('company') can be employed depending on (but not limited to) the following factors:

- level of management's maturity (time- versus quality-driven approach),
- level of QA's maturity (staff, test tools, procedures), and
- whether the company elects to release the software with or without simultaneous troubleshooting.

Among the many interfaces possible, let us focus here on two. Both are popular, but differ in the way they place QA activities in the life cycle:

(1) *Incremental (gradual) interface*—the software is turned over to QA a part at a time (i.e., some pieces of a new functionality and/or fixes as they become ready for QA attention).
(2) *All-in-one interface*—the entire software is turned over to QA once only, usually shortly before the release of the software to production.

One can anticipate a wide range of benefits and drawbacks inherent in the implementation of each interface. When the incremental interface is employed, for example, one can expect the following benefits:

- quality assurance analysts can focus on a new piece of functionality at a time,
- maintenance management has more flexibility to assign resources,
- a functionality's implementation is more controllable (i.e., outstanding bugs can be fixed per functionality), and
- parts of the software may be retested many times.

When the all-in-one interface is employed, one can expect another set of benefits:

- quality assurance analysts have the opportunity to test the entire new and changed functionality of the software,
- test scripting can be tailored for the entire new and changed functionality from the very beginning and then tuned up along the way, and
- testing does not have to be repeated several times.

In evaluating these interfaces and concentrating our attention on the implementation phase, one should keep in mind that the more the implementation phase is broken down into parallel series of stages, the greater is the number of turnovers to QA required by some SDLC/SMLC. With those additional parallel series may come more unit testing and integration testing. Most importantly, regression testing and functional testing often have to be repeated to confirm the continued integrity of what was previously tested.

Since at least one software turnover is always a part of any SDLC and/or SMLC with QA, some QA expenses are unavoidable. At the same time, every software turnover always costs the company time and money, increases time to the market, and reduces profit, without always adding improved customer or company confidence in the software quality. Extra costs *are* avoidable.

Our focus in this paper is on the QA cost and time for testing starting with the turnover to QA just before the release of the software for customer use. Depending on what QA work was done in the earlier phases of the SDLC/SMLC, two different approaches can be considered for the two QA interfaces:

- *a simplified approach*—the entire set of testing actions has to be repeated after every software turnover,
- *a more realistic approach*—only a subset of testing actions has to be repeated after every software turnover.

This distinction will help us more realistically compute the time and cost difference between the two interfaces.

3. CALCULATIONS

3.1. Simplified formulas

3.1.1. Incremental interface

For the incremental interface, let us consider the technological chain starting from the point where (a part of) the software is turned over to QA. Typically, the module carrying out the changes is tested first; then the entire software is regression tested to be sure that these changes do not impair the entire software's behaviour.

Let us call the amount of the time required for complete testing of the module that has been modified Q_1, and time for regression and complete testing of the entire package Q_0. Thus, the complete time Q_{01} required for testing of the software with one piece of a new/updated functionality is equal to the sum:

$$Q_{01} = Q_0 + Q_1 \tag{1}$$

When another piece of a new/updated functionality arrives, the module carrying out this functionality gets tested first. The functionality changed in a previous step (since it cannot be stable enough by definition) should be tested next, and then, the entire software. Thus, the additional time Q_2 is required.

The complete time Q_{02} required for testing of a product with two pieces of a new/updated functionality is thereby equal to the sum:

$$Q_{02} = Q_{01} + Q_2 = Q_0 + Q_1 + Q_2 \tag{2}$$

The two steps together will require the testing time T_2:

$$T_2 = Q_{01} + Q_{02} = 2Q_0 + 2Q_1 + Q_2 \tag{3}$$

If we have to test the third, the fourth, etc. piece of a newly added/changed functionality, some more time will be required.

For k turnovers, from the definition QA time T_k is equal to:

$$T_k = \sum_{j=1}^{k} \sum_{i=0}^{j} Q_i \tag{4}$$

We can get a cleaner formula by rearranging the terms collecting all Q_i together for each i. All k inner sums have a Q_0 and a Q_1, so the coefficients to these are both k. As for the other Q_i, there is one less, or $k - 1$, Q_2's and, again, one less, or $k - 2$, Q_3's, etc. This works right down to Q_k of which there is only one, or $(k + 1 - k)$. The general coefficient is $(k + 1 - i)$ and the general sum is:

$$T_k = kQ_0 + \sum_{i=1}^{k} (k + 1 - i)\, Q_i \tag{5}$$

3.1.2. All-in-one interface

Beginning our evaluation at the same point of the procedural chain (when the software is turned over to QA), we should first realize that all of the added/changed functionality and then the entire new product should be tested. In the case of turning over two pieces of new/updated functionality (functional turnovers) the following testing time will be required (see equation (2) above):

$$t_2 = Q_0 + Q_1 + Q_2 \tag{6}$$

Thus, in the general case, for k turnovers, QA time equals:

$$t_k = \sum_{i=0}^{k} Q_i \tag{7}$$

3.1.3. Time difference

To calculate the time difference (ΔT_k) between two interfaces for k turnovers ($k \geqslant 2$), we can simply subtract the amount of QA time required by the all-in-one technology (t_k) shown in equation (7) from the amount of QA time required by the incremental technology (T_k) shown in equation (4):

$$\Delta T_k = T_k - t_k = \sum_{j=1}^{k} \sum_{i=0}^{j} Q_i - \sum_{i=0}^{k} Q_i = \sum_{j=1}^{k-1} \sum_{i=0}^{j} Q_i = T_{k-1} \tag{8}$$

3.1.4. Cost difference

Time is certainly money, but to what degree? Let us calculate, for example, the time difference for $k = 3$ (three turnovers; see equation (3)):

$$\Delta T_3 = T_2 = 2Q_0 + 2Q_1 + Q_2 \tag{9}$$

Assuming that the testing includes writing test plans and making a hardware/software setup prior to actual testing, let $Q_0 = 480$ hours (12 person/weeks) and $Q_1 = Q_2 = 80$ hours (2 person/weeks).

Then $\Delta T_3 = 1\,200$ hours (30 person/weeks). Assuming the average annual cost of an employee equal to \$52K, the incremental technology costs a company an additional \$30K for just three turnovers which comes to \$10K per turnover.

3.2. More realistic formulas

3.2.1. Repeated testing

One does not have to spend the same amount of time testing the same piece of software over and over. Consider a typical situation in which the test plan/test script for testing the previous step has already been developed; hardware and software for testing purposes have been selected and used; and QA analysts have developed an understanding of the functionality under testing. Then we can decrease the amount of time to retest this step m times. To make the calculation simpler, we will assume that m is constant from one turnover to another.

3.2.2. Incremental interface

Let us again denote the amount of the time required for testing, first for the module carrying out the new/updated functionality and then for the regression testing of the entire software, to be sure that the changes do not make the entire software's behaviour less acceptable—that is, Q_1 and Q_0 accordingly. Thus, the complete testing time required for this Q_{01}^m is equal to the sum (see equation (1)):

$$Q_{01}^m = Q_0 + Q_1 \tag{10}$$

When another new/updated functionality arrives, the module carrying out the changes should be tested first. Thus, the additional time Q_2 is needed. Then the changes made in a previous step (since it cannot be stable enough by definition to avoid the need for testing it) should be tested, followed by testing the entire software. Since we assumed we can decrease the amount of time to retest the previously tested step m times, the complete testing time required Q_{02}^m is in this instance a modification of equation (2) for two steps:

$$Q_{02}^m = \frac{1}{m} Q_{01}^m + Q_2 = \frac{1}{m}(Q_0 + Q_1) + Q_2 \tag{11}$$

The two steps together require the time T_2^m (see equation (3)):

$$T_2^m = Q_{01}^m + Q_{02}^m = \frac{m+1}{m}(Q_0 + Q_1) + Q_2 \tag{12}$$

When additional new/updated functionality gets turned over, first the module carrying out this functionality should be tested. Thus, the additional testing time Q_3 is needed. Then the changes made in the previous steps should be again tested, followed by regression testing of the entire software. Then the complete time required Q_{03}^m is equal to the sum:

$$Q_{03}^m = \frac{1}{m}\left(\frac{1}{m}(Q_0 + Q_1) + Q_2\right) + Q_3 \tag{13}$$

All steps together require the testing time T_3^m:

$$\begin{aligned} T_3^m = T_2^m + Q_{03}^m &= \frac{m+1}{m}(Q_0 + Q_1) + Q_2 + \frac{1}{m}\left(\frac{1}{m}(Q_0 + Q_1) + Q_2\right) + Q_3 \\ &= \frac{m^2 + m + 1}{m^2}(Q_0 + Q_1) + \frac{m+1}{m} Q_2 + Q_3 \end{aligned} \tag{14}$$

In the general case, for k turnovers, QA time T_k^m is equal to:

$$T_k^m = \sum_{j=1}^{k} \sum_{i=0}^{j} \alpha_{ij} Q_i \tag{15}$$

for some α_{ij}. The description tells us that for a given Q_i the coefficients α_{ij} must have exactly one 1, exactly one $1/m$, exactly one $1/m^2$, and so on for as many times as Q_i appears. For Q_0 this is k and the coefficient is:

$$\sum_{j=0}^{k-1} \left(\frac{1}{m}\right)^j = \frac{\frac{1}{m^k} - 1}{\frac{1}{m} - 1} = \frac{1}{m-1}\left(\frac{m^k - 1}{m^{k-1}}\right) \tag{16}$$

For all other i, the derivation of T_k shows the coefficient has $k - i + 1$ terms. Applying the same reasoning as for $i = 0$ to the different terms we get

$$T_k^m = \frac{1}{m-1}\left(\frac{m^k - 1}{m^{k-1}} Q_0 + \sum_{i=1}^{k} \frac{m^{k-i+1} - 1}{m^{k-i}} Q_i\right) \tag{17}$$

3.2.3. All-in-one interface

When all the changes made are submitted at once, the process has fewer steps. In the case of two new/updated pieces of functionality (in fact, potential functional turnovers) the following testing time will be required (see equation (6) but including the m factor):

$$t_2^m = Q_0 + Q_1 + Q_2 \tag{18}$$

Then in the general case for k turnovers (see equation (7), QA time will be equal to:

$$t_k^m = \sum_{i=0}^{k} Q_i \tag{19}$$

3.2.4. Time difference

To calculate the time difference (ΔT_k^m) between two procedures for k turnovers ($k \geqslant 2$), we can simply subtract the amount of QA time required by the all-in-one interface (t_k^m, equation (19)) from the amount of QA time required by the incremental interface (T_k^m, equation (15)):

$$\Delta T_k^m = T_k^m - t_k^m \tag{20}$$

In other words, ΔT_k^m is T_k^m or $\sum_{j=1}^{k} \sum_{i=0}^{j} \alpha_{ij} Q_i$ with all the α_{ij} that are 1 removed. This means the amount can be calculated with the alternate formula for a geometric progression and is equal to:

$$\Delta T_k^m = \frac{\frac{1}{m^k} - \frac{1}{m}}{\frac{1}{m} - 1} Q_0 + \sum_{i=1}^{k} \frac{\frac{1}{m^{k-i+1}} - \frac{1}{m}}{\frac{1}{m} - 1} Q_i = \frac{1}{m-1}\left(\left(1 - \frac{1}{m^{k-1}}\right) Q_0 + \sum_{i=1}^{k} \left(1 - \frac{1}{m^{k-i}}\right) Q_i\right) \tag{21}$$

3.2.5. Cost difference

Let us calculate, for example, the time difference for $k = 3$ (three turnovers) and $m = 2$ (the amount of time to retest is only half of that required for initial testing). Then equation (21) simplifies to:

$$\Delta T_3^2 = \frac{3}{4}(Q_0 + Q_1) + \frac{1}{2}Q_2 \tag{22}$$

Assuming again that the testing includes writing test plans and making a hardware/software setup prior to actual testing, let Q_0 = 480 hours (12 person/weeks) and $Q_1 = Q_2 = 80$ hours (2 person/weeks). Then $\Delta T_3 = 460$ hours (11·5 person/weeks). Assuming the average annual cost of an employee is equal to \$52K, the incremental technology costs a company an additional \$11·5K for just three turnovers which comes to \$3·8K per turnover per version of the software. These figures may not appear to be very large, but it is what can be saved *per turnover per version of the software.* If we require five turnovers per product and release five products/versions a year, then we save \$95K a year!

3.3. Lost and found

Where there are extraneous expenditures, there is almost always the possibility of savings. Most companies do not realize the degree to which the technologies and processes they employ, correlate with profitability. The result is that these companies are sacrificing profit for the convenience and convention of maintaining their technology and process status quo, not for maintaining their software. As strange as it may sound, this is especially true of small and medium-sized companies (up to 100 employees in Information Systems). Apparently in swearing by the adage, 'If it ain't broke, don't fix it,' these companies are collectively losing anywhere from hundreds of thousands to millions of dollars a year.

The situation becomes even more complicated when a company maintains software on multiple platforms, and when a company runs several maintenance efforts concurrently on the same software. Each project and sub-project proceeds at its own pace and is at a different place in its own life cycle. The result is the potential for uncoordinated and multiple turnovers to QA. Such turnovers could lead to a compounded set of unnecessary SDLC/SMLC expenses, not to mention an organizational morass and needless stress. Then invariably, software of lesser quality gets manufactured at a greater expense than necessary.

Let us compute the time differences between eight and two turnovers using the formulas previously derived. Since the formulas compare k and one turnovers, we first need to calculate separately ΔT_8 and ΔT_2. We will further assume that $m = 2$; what this means is that we will need only half the time to retest the same piece of software because part of whatever is necessary for retesting (test plan/test script, hardware setup, testing tools application, etc.), has already been developed in previous testing.

Another assumption concerns the required number of testing hours and money involved. Assuming as before that testing includes writing test plans, etc., prior to actual testing, let one more time Q_0 = 480 hours (12 person/weeks) and $Q_1 = Q_2 = \ldots = Q_8 = 80$ hours (2 person/weeks). Then ΔT = 676 hours (16·9 person/weeks). The average annual cost of an employee could be considered \$52K. Then incremental interface costs a company an additional \$16·9K for just six extra turnovers which comes to *\$2·8K per turnover per version of the software.*

4. REALITY CHECK

Running well-designed full-bore validation tests at low overhead cost in actual company environments is nearly impossible. A literature search was unproductive in finding a cost comparison for the use of the all-in-one and incremental interfaces (e.g., Phister, 1979, pages 504–505). As a substitute reality check, a search was made for a company that had cost experience data available for the incremental interface. The search turned up in the Boston–Washington corridor a medium-size company supporting multiple software product lines.

A experiment was set up and run at the company site on the maintenance work on one product line. The company's usual cost data collection and management processes were used, except for the substitution of the all-in-one for the incremental interface. The turnovers studied were the final ones that would have immediately been followed by the release to the customers of the new/revised version. A comparison of the time and money spent in incremental turnovers of a similar product for similar kinds of changes in the past, found a difference of about $10K for five extra turnovers. That average came to about *$2K per turnover per version of the software.* Thus, the turnover cost savings for the company actually amounted to a substantial and significant reduction in overall maintenance costs, though over 28 per cent less than the formulas predicted. For a single experiment, this experimental result seems reasonably close to the predicted result. It is hard to determine precisely why savings did not amount to their full potential, and the contribution of each factor to the cost savings. However, in this experiment, one could speculate from the data that parameter m could not be estimated sufficiently precisely (Dobbins and Buck, 1987).

5. DISCUSSION

The experimental support for the message from the formulas suggests a motto for cost-effective management of the turnover to QA: '*Live Short and Prosper*'! The situation is one where less is more. That is, consolidating the incremental turnovers into a single turnover saves time and money for the company. Can we not argue that a few simple formulas showed the way to saving time and money by moving to an all-in-one turnover interface?

On the negative side, such action is sure to meet with opposition in most companies. Software maintainers and developers do an impossible job by an even more impossible deadline. Yet, at the end of the long arduous road, is some QA manager—the 'software maintenance and development police officer', so to speak—telling the maintainer or developer how to do his or her job better! That does not fly well.

So how do QA personnel work and play well with others? The answers are the stuff of future papers. One important clue to the answer, however, bears particularly on 'Live *Short* and Prosper:' self-policing. No methodology is perfect, but that does not mean we should not ask why. Why were the actual savings 28 per cent removed from the anticipated ones? What factors, besides those already identified, should be recognized, and how? What can we do, in other words, to 'purify' our development, maintenance, and QA environments so that the customers, the employees and the company can all prosper more?

Among the more specific factors may be some of these. This paper concentrated on the turnover just before version release, and concentrated on testing. Many life cycles ask

for QA participation at earlier or many points in the life cycle, and many involve activities other than test runs done by QA personnel—for example, design inspections, functional verifications, walk throughs, management reviews and customer-conducted beta tests. Also, the purity and thoroughness of any of the QA activities has not been addressed—for instance, how much testing is enough? The results of exploring such topics are beyond the scope of this paper.

As QA professionals we are obliged to keep asking questions and changing what we do until we get it right. What is right is not only a moving target, but also involves deciding about for whom and when. After all, we and our customers demand nothing less than continued improvement of our software; we and the companies we work for, cannot afford to demand anything less of ourselves.

6. CONCLUSION

This paper provides support for using the all-in-one interface in preference to the incremental interface to quality assurance (QA). The place in the SDLC/SMLC focused on comes just before the release of a version of the software, and typically is the last phase in the life cycle devoted to testing the software. That testing can be done incrementally as parts of the software become ready, or can be done all at one time by accumulating the parts until all are ready for testing. When action alternatives exist, some simple calculations may be of help in choosing the action.

This paper derived simple formulas for estimating the time and costs associated with each of the two interfaces. Those formulas indicated a lower cost and shorter time for the use of the all-in-one interface. An experiment with trying the all-in-one interface in a company that normally used the incremental interface, gave results reasonably consistent with what the formulas predicted. These results suggest the motto 'Live *Short* and Prosper' as a way of emphasizing the financial benefits for the company by adopting the all-in-one interface with QA as part of the company's software life cycles.

Acknowledgements

I am indebted to and thank the anonymous reviewers for their constructive comments.

References

Arthur, L. J. (1988) *Software Evolution*. John Wiley & Sons, Inc., New York, NY.

Boehm, B. (1981) *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

Chapin, N. (1988) 'Software maintenance life cycle', in *Proceedings of the Conference on Software Maintenance—1988*, Phoenix, AZ, IEEE Computer Society Press, Los Alamitos, CA, pp. 6–13.

Dobbins, J. H. and Buck, R. D. (1987) 'The cost of software quality', in Schulmeyer, G. G. and McManus, J. I. (Eds), *Handbook of Software Quality Assurance*, Van Nostrand Reinhold, New York, NY, pp. 119–163.

Harrold, M. J. and Soffa, M. L. (1988) 'An incremental approach to unit testing during maintenance', in *Proceedings of the Conference on Software Maintenance—1988*, Phoenix, AZ, IEEE Computer Society Press, Los Alamitos, CA, pp. 362–367.

Martin, J. and McClure, C. L. (1983) *Software Maintenance*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

McManus, J. I. (1987) 'How does software quality assurance fit in?' in Schulmeyer, G. G. and McManus, J. I. (Eds), *Handbook of Software Quality Assurance*, Van Nostrand Reinhold, New York, NY, pp. 14–24.

Osborne, W. M. (1985) *Executive Guide to Software Maintenance*, National Bureau of Standards

Publication 500-130, Washington DC, 28 pp., reprinted in Longstreet, D. H. (Ed) (1990) *Software Maintenance and Computers*, IEEE Computer Society Press, Los Alamitos, CA, pp. 2–13.
Phister, M. (1979) *Data Processing Technology and Economics*, Second Edition, Digital Press, Bedford, MA.
Rothermel, G. and Harrold, M. J. (1993) 'A safe, efficient algorithm for regression test selection,' in *Proceedings of the Conference on Software Maintenance—1993*, Montreal, Quebec, IEEE Computer Society Press, Los Alamitos, CA, pp. 358–367.

Author's biography:

Vladimir M. Tsivkin serves as the Director of Software Quality Assurance for Ex Machina, Inc., in New York, NY. Before coming to the USA, he worked in Russia as a systems analyst and software professional. Vladimir has authored a number of published papers in operations research and management science applications. His interests include reliability modelling, software measurements and metrics, costs versus quality analysis, and business-orientated software life cycle modelling. Vladimir earned his M.S. degree in Electrical Engineering from the Electrotechnical Institute in St. Petersburg, Russia. He is a member of the American Society for Quality Control, the Institute of Electrical and Electronic Engineers, and the New York Academy of Sciences. His e-mail address is: vmtsivkin@aol.com